# Data Structures for Moving Objects on Fixed Networks

by

Thuy Thi Thu Le and Bradford G. Nickerson

TR06-181, February 13, 2007

Faculty of Computer Science

University of New Brunswick

Fredericton, N.B. E3B 5A3

Canada

Phone: (506) 453-4566

Fax: (506) 453-3566

Email: fcs@unb.ca

www: http://www.cs.unb.ca

# 1. Introduction

Unlike spatial databases [9] containing stationary spatial objects, (e.g., gas stations, rivers, cities, roads, regions), spatio-temporal databases [3] contain spatial objects that move over time (e.g., moving cars, moving people, clouds, flights). In last few decades, both spatial and spatio-temporal databases have received much attention from researchers. One of the biggest challenges in spatial and spatio-temporal databases is how to improve the response time for query processing of moving objects, so called continuous query processing [10]. One of queries related to moving objects is, for example, *"how many moving cars are in the center of Fredericton in the time interval* $(t1, t2)$ *?"*

In order to answer this kind of query efficiently, many indexes structured for capturing moving objects have been introduced, such as the R*-tree [2], MON-tree [5], TPR-tree, and FNR-tree [6]. While moving objects captured by the R*-tree and TPR-tree are not constrained in any motion model, the FNR-tree and MON-tree are two new indexes, which are especially used for moving objects in the road network. In [5], the MON-tree is claimed to be two to four times faster than the FNR-tree.

In this report, the necessary background for the MON-tree, which is related to the index structure, the insertion algorithm and the searching algorithms, is described in Section 2. Section 3 then shows our experiments after we implemented the MON-tree as well as our comparison results between the MON-tree and the R*-tree implemented for rectangles. In Section 3, we also discuss the way we generated test data from the real Canadian road network [1]. Finally, the conclusion of our report is shown.

# 2. Background of the MON-tree

## 2.1 Index Structure

The MON-tree stands for a tree for Moving Objects in Networks. This data structure is based on the R-tree index structure [7]. In order to capture moving objects on road networks, the data structure of the MON-tree contains three main parts. First, a top R-tree is used to capture the road network. Second, a set of bottom R-trees, each of which is connected by leaf nodes of the top R-tree, are used to represent moving objects on the road. Finally, a hash table is used to organise these top and bottom R-trees efficiently. Figure 1 shows an example of a road network. There are eight roads (e.g., $e1, e2$,..., and $e8$) and eleven moving objects in this road network. Figure 2 shows the data structure of the MON-tree representing these moving objects on the roads.

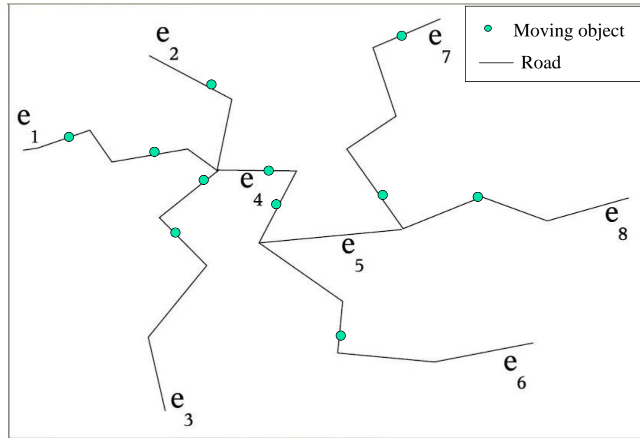


Figure 1: Example of a road network [5].

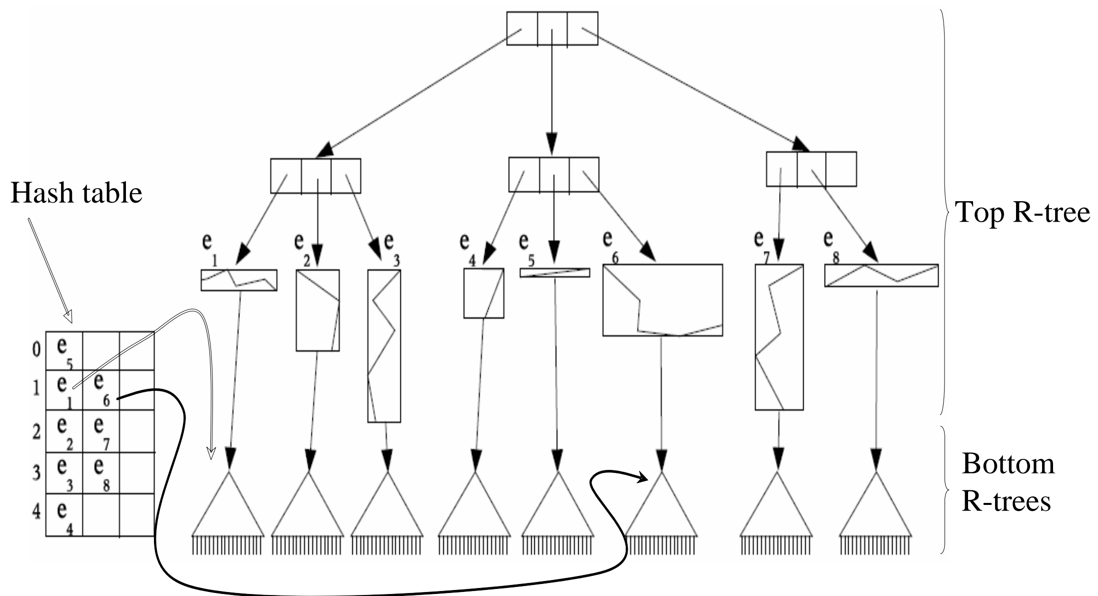


Figure 2: Data structure of MON-tree [5].

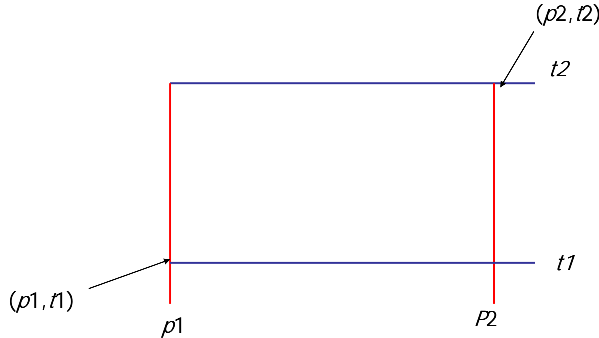


Figure 3: A rectangle representing time and positional interval.

### a. Top R-tree

Like the conventional R-tree, the top R-tree of the MON-tree (see Figure 2) indexes the bounding box of polylines. The bounding box for each leaf is the smallest box that spatially contains a polyline (i.e., a road) of the road network, while the bounding box of a non-leaf node is the smallest box that contains the bounding boxes of the child nodes.

### b. Bottom R-trees

For movement representation, each bottom R-tree is used to index moving objects in each road, which is indexed by the top R-tree. In order to represent a moving object (e.g., its location, its velocity, etc.), a time interval $(t1, t2)$ and a position interval $(p1, p2)$ are necessary for each object. $p1, p2$ is the relative position of the objects inside the polyline at times $t1$ and $t2$, respectively and $0 \leq p1, p2 \leq 1$. This time interval and position interval together are represented as a rectangle with two vertices $(p1, t1)$ and $(p2, t2)$ as in Figure 3.

In a bottom R-tree, each node contains an identification of an object (e.g., *oid*). If the current node is a leaf node, it contains a rectangle formed by two vertices $(p1, t1)$ and $(p2, t2)$. However, if it is an internal node, it keeps a maximum bounding box (MBB) of all MBBs of time and position intervals of its child nodes. Since each node of a bottom R-tree is actually a rectangle, we can use R*-tree algorithms for a set of rectangles [2] to insert moving objects (i.e, a rectangle) into the bottom tree.

### c. Hash table

The hash table is organized by the identification of the road (i.e., *polyid*). In addition, each element of the hash table contains a link to a bottom R-tree. The purpose of the hash table is to organize data in the top tree and bottom trees efficiently. Since a bottom tree is pointed to by both the hash table and the top tree, we can check the presence of

objects in a road through the hash table, without using the top tree. By doing this, if a road does not have any moving object in it, this road does not need to be inserted into the top tree. By avoiding the storage of polylines without moving objects in this way, we can keep the top tree as small as possible, thus reducing the search time of the query in the top tree.

Like the hash table, the top tree also points to bottom trees. However, the top tree is useful for window queries to find which polylines (i.e., roads) intersect the query window.

## 2.2 Insertion Algorithm

### a. Polyline Insertion

To avoid having polylines without moving objects in the top R-tree, the insertion of a polyline does not actually happen until this polyline has the first moving object in it. Thus, given a new road, we first initialize a *polyid* with a null pointer in the hash table. We start to insert it whenever we need to insert the first object of this polyline to a bottom tree.

### b. Moving Object Insertion

A moving object with an object identification (i.e., *oid*), polyline identification (i.e., *polyid*), a position interval $p = (p1, p2)$ and a time interval $t = (t1, t2)$ are inserted into the tree. The insert algorithm follows three main steps:

- A search for the polyline (*polyid*) in the hash table
- If the bottom R-tree of the polyline is NULL
  - the polyline's MBB is inserted to the top R-tree
  - the rectangle $(p1, p2, t1, t2)$ is inserted to the corresponding bottom R-tree
  - the leaf node in the top R-tree and Hash table to the bottom R-Tree are linked
- Else: the rectangle $(p1, p2, t1, t2)$ is inserted to the corresponding bottom R-tree

Consider the following algorithm for building the MON-tree. The call $B\text{UILD}T\text{REE}(T, E, n)$ builds the MON-tree whose root is pointed by $T$. $E$ is an array containing data of $n$ roads on the networks. The INSERTBOTTOM algorithm creates the bottom R-tree pointed to $hash_i.bottomTree$. The INSERTTOP algorithm inserts a road $E[i]$ into the top R-tree.

```
BuildTree(T, E, n)
 1  oid ← 0
 2  for (i = 0; i < n; i ← i + 1)
 3  do m ← a random number ∈ {0, .., max} of moving objects in each road
 4      if (m ≠ 0)
 5         then pp ← m pairs of random position intervals
 6              pt ← m pairs of random time intervals
 7              hash_i ← the element of the hash table corresponding to road E[i]
 8              while (pp ≠ NULL)
 9              do if (hash_i.bottomTree = NULL)
10                    then insertBottom(hash_i.bottomTree, pp, pt, oid)
11                         insertTop(T, E[i], hash_i.bottomTree)
12                         oid ← oid + 1
13                    else insertBottom(hash_i.bottomTree, pp, pt, oid)
14                         oid ← oid + 1
15                 pp ← pp.NextPtr
16                 pt ← pt.NextPtr
```

Figure 4: The algorithm for building a MON-tree.

## 2.3. Searching Algorithm

Like the query example discussed in Section1 *"how many moving cars are in the center of Fredericton in the time interval* $(t1, t2)$ *?"*, a query window $Q = (x1, x2, y1, y2, t1, t2)$ will find all objects within the area $Q_w = (x1, x2, y1, y2)$ during the time interval $Q_t = (t1, t2)$. Figure 5 shows an example of the window query, which intersects six roads ($e1$, $e2$, $e3$, $e4$, $e5$, and $e6$) and has five objects in range.

In order to answer the window query, the searching algorithm of the MON-tree is performed on the top tree and the bottom trees as follows:

1. The polylines' MBBs that intersect the query rectangle $Q_w$ in the top R-tree are found.

2. In each polyline, the intervals where the polyline intersects $Q_w$ are found using the real polyline representation. Figure 6 shows an example where $Q_w$ intersects three roads ($e7$, $e8$, and $e5$). While intersecting road $e7$, this query rectangle divides $e7$ into three position intervals lying inside the query rectangle such as (0, 0.1), (0.6, 0.7), and (0.8, 1). These three intervals are combined with the time interval $Q_t = (t1, t2)$ to form three sub-query windows for the bottom R-tree of the road $e7$ (see the middle part of Figure 6).

3. Given a set of window queries $Q'$ found in the second step for each polyline, where

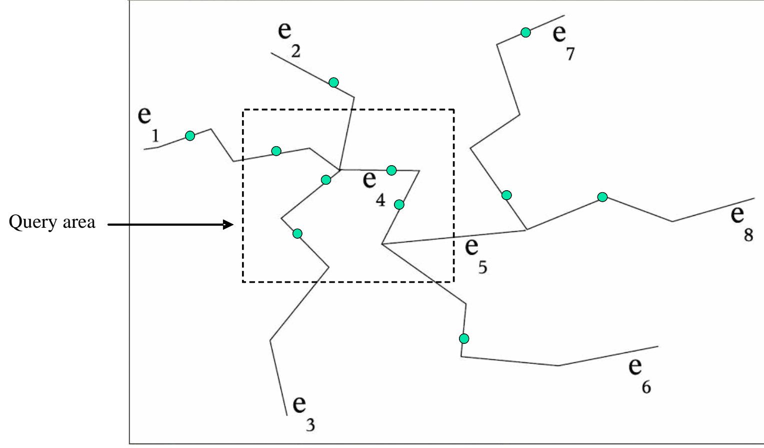


Figure 5: Example of a window query.

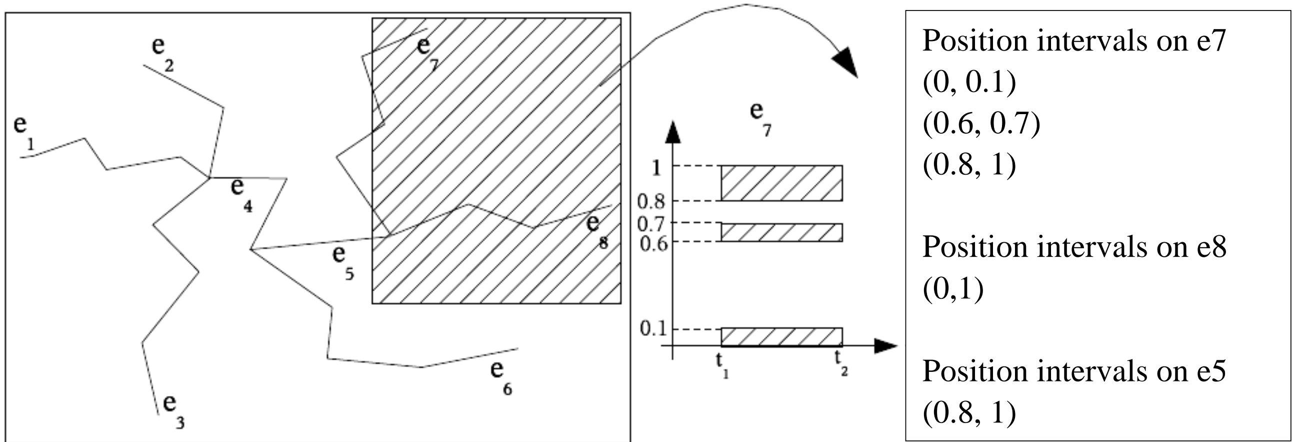


Figure 6: Example of polyline intersected intervals.

$Q' =$ (position intervals, a time interval), the search algorithm now goes down to the corresponding bottom tree of the polyline and searches for moving objects falling in one of the rectangles in $Q'$. The MBB of a node in the bottom tree satisfies $Q'$ if it intersects with at least one sub-window in $Q'$.

For more detail, we used Figure 7 and Figure 8 to show the search algorithm for the MON-tree. In these figures, the algorithms are described by a pseudo code. In Figure 7, the algorithm, so called $\textsc{Searching}(T, E, Q_w, Q_t)$, has its inputs including a pointer $T$ to the root of the MON-tree, an array $E$ containing the real representation of roads, a rectangle query $Q_w$, and a time interval query $Q_t$. Corresponding to the first and second steps above, this algorithm repeatedly finds nodes whose MBBs intersect to $Q_w$ (see lines 4 to 7) until a leaf node is reached. When this situation happens (line 1), a list of position

intervals formed by intersections of roads and $Q_w$ is searched (line 2) and the algorithm SEARCHBOTTOMTREE (line 3) is called to continue with searching on the corresponding bottom MON-tree.

In Figure 8, the algorithm SEARCHBOTTOMTREE$(T, Q_{ws}, Q_t)$ has its inputs including a pointer $T$ to the root of a bottom MON-tree, and a list of query rectangles formed by a set of position interval $Q_{ws}$, and a time interval $Q_t$. As discussed in the third step above, the MBB of a node of $T$ is satisfied if it intersects with at least one query rectangle formed by $Q_{ws}$ and $Q_t$. Towards this idea, lines 3 to 13 are recursively called to check if a descendant of $T$ intersects with at least one query rectangle. When a descendant of $T$ is reached to a leaf node of the bottom MON-tree (line 1), the corresponding moving object of this leaf node is counted as an object in range (line 2).

# 3. Experiments

## 3.1. Implementation

In this report the MON-tree is implemented using C++. Besides conventional algorithms the R-trees, such as inserting nodes, splitting a node, checking the least enlargement of a bounding box when choosing the inserted node, the MON-tree algorithms also work with a hash table and several R-trees (e.g., a top tree and bottom trees) when building a MON-tree and deal with the polyline intersection problem when processing query windows.

In order to compare the MON-tree and R*-tree, we use the R*-tree as a 3D R-tree to represent moving objects. A leaf node of this R*-tree contains a two-dimensional rectangle for a real position interval of an object and one additional dimension for the time interval of the object. The R*-tree source code for a set of rectangles [12] is re-used. However, some modifications on this source code are made to ensure that the program will run

SEARCHING$(T, E, Q_w, Q_t)$
1 **if** $(T.attribute = LEAF)$
2 **then** $Q_{ws} \leftarrow$ a list of road intervals created by intersections of roads and $Q_w$
3 searchBOTTOMTREE$(T.bottomTree, Q_{ws}, Q_t)$
4 **else** $n \leftarrow$ number of child nodes of T
5 **for** $(i = 0; i < n; i \leftarrow i + 1)$
6 **do if** $(\text{INTERSECT}(T.child[i], Q_w) = TRUE)$
7 **then** SEARCHING$(T.child[i], Q_w, Q_t)$

Figure 7: The algorithm for searching on the top MON-tree.

```
SEARCHBOTTOMTREE(T, Q_ws, Q_t)
 1  if (T.attribute = LEAF)
 2     then return  one moving object in range
 3     else  n ← number of child nodes of T
 4           for (i = 0; i < n; i ← i + 1)
 5           do p_w ← Q_ws
 6              flag ← FALSE
 7              while (p_w ≠ NULL)
 8              do
 9                  if (INTERSECT(T.child[i], p_w) = TRUE)
10                     then flag ← TRUE
11                  p_w ← p_w.NextPtr
12              if (flag = TRUE)
13                 then SEARCHBOTTOMTREE(T.child[i], Q_ws, Q_t)
```

Figure 8: The algorithm for searching on a bottom MON-tree.

properly with our generated test data.

Since we only focus on moving objects (e.g., vehicles) on a road network, we use the New Brunswick road network (see Figure 10), real road data from a real Canadian road network [1] for testing. This data file contains 65688 roads (i.e., polylines). For the MON-tree, all test data are prepared as follows:

1. We first generate data of the New Brunswick road network from the Canadian road network. Since the original GML data file of the New Brunswick road network contains some information on roads, such as road identifications, road classes, addresses of roads, and coordinates of roads, we transformed this GML data file into a smaller text file, which contains only coordinates of polylines (i.e., roads). In the Appendix, we show an example of the input GML file, and its transformed text file.

2. Based on the road data, we then randomly generate data (normalized position intervals $(p1, p2)$) of moving objects on roads. We assumed that in each road, the velocity $v$ of moving objects is the same. In addition, each road is limited to a specific velocity, which is randomly generated between a maximum velocity $V_{max}$ (e.g., 100 km/h) and a minimum velocity $V_{min}$ (e.g., 10 km/h).

3. Finally, we generate time intervals $[t1, t2]$ for moving objects. Since a distance $\Delta s = v \times \Delta t$, we compute $\Delta t = (t1 - t2) = \Delta s \div v = (p2 - p1) \div v$. A time interval is generated by assigning $t1$=0 and $t2 = \Delta t$. Figure 9 illustrates a randomly generated moving object with velocity 10 km/h.

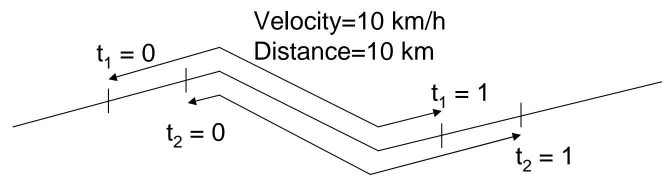


Figure 9: Example of two randomly generated moving objects

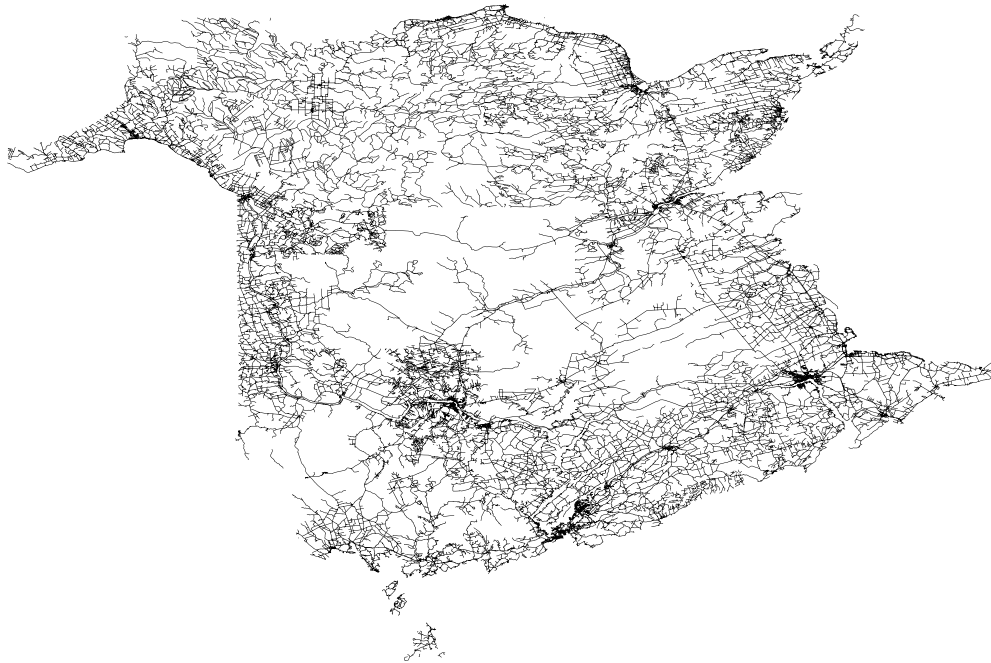

Figure 10: The road network of New Brunswick containing 65,688 roads. This picture was drawn using the TatukGIS Viewer open source tool.

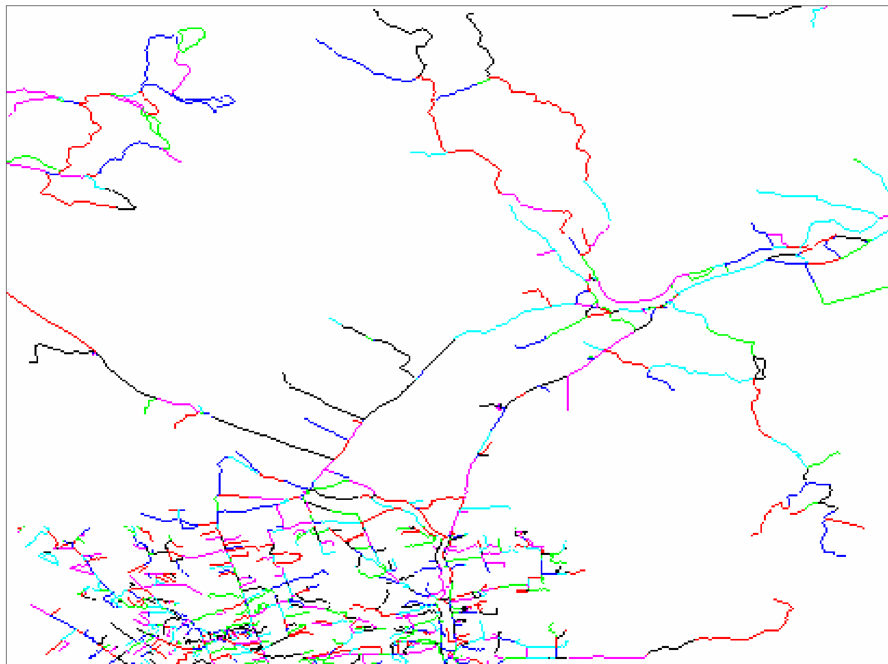

Figure 11: An area of the New Brunswick road network. Each road is drawn in a different color using a custom openGL program.

In order to compare the MON-tree to the R*-tree, similar test data must be generated for the R*-tree. Unlike the MON-tree, which indexes a road network and moving objects on it separately, the R*-tree indexes moving objects, their position intervals on the road network, and time intervals together. The test data for the MON-tree are transformed into test data for the R*-tree as follows:

1. Data of objects (real position intervals) from the test road data and the moving object data of the MON-tree are computed.

2. The random time intervals for moving objects used for the MON-tree testing are reused.

Furthermore, we also compare the MON-tree with a naïve linear search. Like the data preparation for the R*-tree, we reuse the same randomly generated data used for building the MON-tree.

## 3.2. Comparing the MON-tree, the R*-tree and a naïve Search

The building tree algorithm and searching algorithm for the MON-tree, the naïve searching algorithm, and the building algorithm of the R*-tree are run on UNIX's quar-

Table 1: Building trees; times (in seconds) and sizes for the MON-tree and the R*-tree. Max objects is the maximum number of moving objects in each road. Note that in all cases, 65688 roads are used. Size is the number of nodes in the tree. The number in parentheses is the ratio of the R*-tree construction time to the MON-tree construction time.

| Max objects | N | MON-tree | | R*-tree | |
|---|---|---|---|---|---|
| | | size | time | size | time |
| 10 | 294,841 | 423,064 | 32.06 | 344,991 | 39.42 (1.23) |
| 20 | 624,736 | 825,086 | 60.14 | 732,290 | 88.94 (1.48) |
| 30 | 956,542 | 1,213,726 | 107.23 | 1,119,660 | 141.22 (1.32) |
| 40 | 1,283,823 | 1,590,992 | 122.21 | 1,501,710 | 191.65 (1.57) |

tet.cs.unb.ca. However, the searching algorithm for the R*-tree is run on UNIX's chorus.cs.unb.ca. The following tables show the running times of these algorithms.

**Building Tree Time**

Based on the R-tree, both the MON-tree and R*-tree are implemented with the same criteria. The maximum (e.g., M) and minimum (e.g., m) numbers of entries in each node are 10 and 5, respectively. In addition, the MON-tree uses 100 buckets in the hash table. Note that MON-tree, the R*-tree and the naïve search are tested using the same set of objects, randomly generated as mentioned in Section 3.1.

From the results shown in Table 1, we see that the time for building the MON-tree is about 1.4 times faster (on average) than that of the R*-tree even though the number of nodes of the MON-tree is about 1.12 times greater (on average) than those of R*-tree.

**Time for searching**

We ran both the trees and the naïve search with the same numbers of moving objects generated from 65,688 roads. The roads and objects are the same in the three tests. We generated 400 random window queries and used them for each test. In our testing, a window query is a square, which is formed by its central point and the length of its edges. Such a random query is generated by three steps. First, we randomly generated the central point of the window query. Second, we generated a random size for the query. Note that the central point of a query must belong to the road network area. Moreover, the size of each window query is up to the width of the road network area of New Brunswick. Finally, a time interval of the query was also randomly generated.

We categorized queries into five types based on the number (F) of moving objects falling in range of their query rectangles, such as $[0, log_2^{1/2}(N))$, $[log_2^{1/2}(N), log_2(N))$, $[log_2(N), log_2^2(N))$, $[log_2^2(N), log_2^3(N))$ and $[log_2^3(N)$, N], called Type 1, Type 2, Type 3,

Table 2: Example of milestones used to categorize queries. N is the number of moving objects.

| N | $log_2^{1/2}(N)$ | $log_2(N)$ | $log_2^2(N)$ | $log_2^3(N)$ |
|---|---|---|---|---|
| 294,841 | 4.26258 | 18.1696 | 330.134 | 5,998.39 |
| 624,736 | 4.38781 | 19.2529 | 370.674 | 7,136.54 |
| 956,542 | 4.45729 | 19.8675 | 394.716 | 7,842.01 |
| 1,283,823 | 4.50467 | 20.292 | 411.766 | 8,355.56 |

Table 3: Average search times (in seconds) and numbers of visited nodes when F $\in$ [0, $log_2^{1/2}(N)$) for the MON-tree, the R*-tree, and the naïve search. h is the number of range searches (out of 400) that were averaged to obtain these results. v is the average number of visited nodes. The number in parentheses is the ratio of the average MON-tree time to this time.

| N | MON-tree | | | R*-tree | | naïve search |
|---|---|---|---|---|---|---|
| | h | time | v | time | v | time |
| 294,841 | 17 | 0.000675 | 26 | 0.0000014 (489.45) | 42 | 0.03412 (0.01980) |
| 624,736 | 24 | 0.000610 | 25 | 0.0000009 (701.53) | 32 | 0.10042 (0.00608) |
| 956,542 | 20 | 0.000002 | 20 | 0.0000005 (3.76) | 24 | 0.1545 (0.00001) |
| 1,283,823 | 17 | 0.000700 | 24 | 0.0000018 (397.96) | 61 | 0.21118 (0.00332) |

Type 4 and Type 5, respectively. Note that N is the total number of moving objects in a tree. Table 2 shows an example of the number of objects falling in these categorized queries. The test harness code is shown in Figure 12.

In order to benchmark the search times of the algorithms, we ran 400 random queries on each set of the generated data for moving objects. For each query we counted the average search time and the number of visited nodes. Five tables Table 3, Table 4, Table 5, Table 6, and Table 7 are used to show the search results for the five types of queries Type 1, Type 2, Type 3, Type 4, and Type 5, respectively. Each of these tables contains the search results for the MON-tree, the R*-tree, and the naïve search.

Table 3 shows that when F $\in$ [0, $log_2^{1/2}(N)$), the average search time of the MON-tree is about 4 to 702 times greater than that of the R*-tree. However, the average search time of the MON-tree is about 51 to 82181 times faster than that of the naïve search.

Similar to results in Table 3, Table 4 shows that when the number of objects in range is small, (e.g., F $\in$ [$log_2^{1/2}(N)$, $log_2(N)$), the average search time of the MON-tree is about 88 to 1815 times greater than that of R*-tree. However, the average search time of the MON-tree is about 23 to 129 times faster than that of the naïve search.

When the number of objects in range is larger, (e.g., F $\in$ [$log_2(N)$, $log_2^2(N)$), Table 5 shows that the R*-tree runs about 320 to 2593 times faster than the MON-tree. In most cases, the MON-tree runs 2 to 10 times faster than the naïve search even though there is

```
GlobalSearching(T, N)
 1  v_1, v_2, v_3, v_4, v_5 ← 0  //number of visited nodes for five types of queries
 2  h_1, h_3, h_3, h_4, h_5 ← 0  //number of queries
 3  time_{h_1}, time_{h_2}, time_{h_3}, time_{h_4}, time_{h_5} ← 0  //total time for executing queries
 4  for (i = 1; i <= N; i ← i + 1)
 5  do center ←  a random point ∈ the rectangle [Xmin, Ymin, Xmax, Ymax]
 6      size_w ←  a random number ∈ [0, Sizemax]
 7      Q_w ←  a square whose center point is center and whose edges' size is size_w
 8      Q_t ←  a random time interval
 9      time_{before} ←  get the current time
10      {F, v} ← Searching(T, Q_w, Qt)
11      time_{after} ←  get the current time
12      time_{total} ← time_{before} − time_{after}
13      switch
14         case 0 ≤ F < log_2^{1/2}(N) :
15               v_1 ← v_1 + v
16               h_1 ← h_1 + 1
17               time_{h_1} ← time_{h_1} + time_{total}
18         case log_2^{1/2}(N) ≤ F < log_2(N) :
19               v_2 ← v_2 + v
20               h_2 ← h_2 + 1
21               time_{h_2} ← time_{h_2} + time_{total}
22         case log_2(N) ≤ F < log_2^2(N) :
23               v_3 ← v_3 + v
24               h_3 ← h_3 + 1
25               time_{h_3} ← time_{h_3} + time_{total}
26         case log_2^2(N) ≤ F < log_2^3(N) :
27               v_4 ← v_4 + v
28               h_4 ← h_4 + 1
29               time_{h_4} ← time_{h_4} + time_{total}
30         case F ≥ log_2^3(N) :
31               v_5 ← v_5 + v
32               h_5 ← h_5 + 1
33               time_{h_5} ← time_{h_5} + time_{total}
34
35  return // {average query time, average number of visited nodes for each type of queries}
36  {time_{h_1} ÷ h_1, v_1 ÷ h_1}
37  {time_{h_2} ÷ h_2, v_2 ÷ h_2}
38  {time_{h_3} ÷ h_3, v_3 ÷ h_3}
39  {time_{h_4} ÷ h_4, v_4 ÷ h_4}
40  {time_{h_5} ÷ h_5, v_5 ÷ h_5}
```

Figure 12: The test harness code for N queries.

Table 4: Average search times (in seconds) and numbers of visited nodes when F $\in [log_2^{1/2}(N), log_2(N))$ for the MON-tree, the R*-tree, and the naïve search. h is the number of range searches (out of 400) that were averaged to obtain these results. v is the average number of visited nodes. The number in parentheses is the ratio of the average MON-tree time to this time.

| N | MON-tree | | | R*-tree | | naïve search |
|---|---|---|---|---|---|---|
| | h | time | v | time | v | time |
| 294,841 | 2 | 0.001518 | 85 | 0.000008 (202.33) | 162 | 0.035 (0.04) |
| 624,736 | 2 | 0.000774 | 117 | 0.000009 (87.68) | 162 | 0.100 (0.01) |
| 956,542 | 2 | 0.006352 | 151 | 0.000004 (1,814.86) | 109 | 0.145 (0.04) |
| 1,283,823 | 2 | 0.001940 | 104 | 0.000016 (121.23) | 346 | 0.210 (0.01) |

Table 5: Average search times (in seconds) and numbers of visited nodes when F $\in [log_2(N), log_2^2(N))$ for the MON-tree, the R*-tree, and the naïve search. h is the number of range searches (out of 400) that were averaged to obtain these results. v is the average number of visited nodes. The number in parentheses is the ratio of the average MON-tree time to this time.

| N | MON-tree | | | R*-tree | | naïve search |
|---|---|---|---|---|---|---|
| | h | time | v | time | v | time |
| 294,841 | 13 | 0.10615 | 864 | 0.000041 (2592.91) | 461 | 0.0354 (3.00) |
| 624,736 | 10 | 0.05555 | 673 | 0.000049 (1133.61) | 430 | 0.0960 (0.58) |
| 956,542 | 12 | 0.02676 | 462 | 0.000055 (482.64) | 600 | 0.1542 (0.17) |
| 1,283,823 | 6 | 0.02167 | 489 | 0.000068 (320.18) | 741 | 0.2150 (0.10) |

Table 6: Average search times (in seconds) and numbers of visited nodes when F $\in [log_2^2(N), log_2^3(N))$ for the MON-tree, the R*-tree, and the naïve search. h is the number of range searches (out of 400) that were averaged to obtain these results. v is the average number of visited nodes. The number in parentheses is the ratio of the average MON-tree time to this time.

| N | MON-tree | | | R*-tree | | naïve search |
|---|---|---|---|---|---|---|
| | h | time | v | time | v | time |
| 294,841 | 58 | 2.7355 | 17,480 | 0.000385 (7,112.26) | 3,377 | 0.0348 (78.54) |
| 624,736 | 56 | 1.3639 | 9,866 | 0.000541 (2,523.27) | 4,031 | 0.1016 (13.42) |
| 956,542 | 67 | 1.0269 | 9,632 | 0.001579 (650.35) | 5,098 | 0.1597 (6.43) |
| 1,283,823 | 48 | 0.6506 | 9,142 | 0.001042 (624.60) | 8,785 | 0.2121 (3.07) |

Table 7: Average search times (in seconds) and numbers of visited nodes when F $\in [log_2^3(N)$, N) for the MON-tree, the R*-tree, and the naïve search. h is the number of range searches (out of 400) that were averaged to obtain these results. v is the average number of visited nodes. The number in parentheses is the ratio of the average MON-tree time to this time.

| N | MON-tree | | | R*-tree | | naïve search |
|---|---|---|---|---|---|---|
| | h | time | v | time | v | time |
| 294,841 | 310 | 30.361 | 179,373 | 0.03256 (932.55) | 155,251 | 0.0440 (689.53) |
| 624,736 | 308 | 21.065 | 166,670 | 0.04722 (446.15) | 219,672 | 0.1151 (182.97) |
| 956,542 | 299 | 20.414 | 205,281 | 0.06217 (328.36) | 318,066 | 0.1803 (113.24) |
| 1,283,823 | 327 | 21.579 | 272,661 | 0.04309 (500.80) | 251,720 | 0.2441 (88.40) |

a case that the average search time of the naïve search is about 3 times faster than that the MON-tree.

When the number of objects in range becomes really large, (e.g., F $\in [log_2^2(N), log_2^3(N))$, or $[log_2^3(N)$, N], both Table 6 and Table 7 show that the R*-tree and the naïve search run faster than the MON-tree. For example, Table 6 show that the R*-tree runs about 625 to 7112 times faster than the MON-tree, also about 328 to 933 times faster than the MON-tree in Table 7. Furthermore, being compared with the MON-tree, the R*-tree runs about 3 to 79 times faster (in Table 6), and about 88 to 689 times faster (in Table 7).

# 4. Discussion

After implementing and comparing the results of the Mon-tree, the R*-tree, and the naïve search, we would like to discuss the following:

### 4.1. What are some reasonable prediction queries for traffic?

In this report, each moving object is indexed by a position interval on a road and a time interval that the object moves on it. A window query is used to ask for moving objects falling in an area (e.g., a rectangle or a square) within a time interval [t1, t2]. Note that the moving time of an object is a certain time in the past up to the present (i.e., the time when the tree is updated); however, the time interval of a query can be in the past, present, or future. If the query is asked for a time interval, which belongs to some objects indexed in the tree, we can search through the tree and find the objects in range. Of course, if the time interval of the query does not exist in the tree (e.g., a time interval in the far past or in the future), there will be no objects in range in this case if we directly search in the tree. However, we can make a prediction about moving objects in the past time or in the future time based on information indexed on the tree. There are some approaches used to predict moving objects [4][13]. An example of a prediction window query is *"How many moving cars will be at a specific section of a road in the next ten minutes?"* A linear movement is considered to simply predict a moving point (i.e., moving objects) [5][13]. Assume an object moves without changing its velocities; the location of the moving object is computed by the following formula: x(t) = x(t0) + v *(t - t0), where x(t0) is the location of the object at a referent time t0; x(t) is the location of the object at a time t; and v is the velocity of the object. Since the values of x(t0), v, t and t0 are known, the value of x(t) is easily computed by the formula. For more accurate results, Jidong Chen et al. considered the situation where a moving object in a road network has many constraints [4], such as the maximum velocity of the road, the number of moving objects moving on that road at the same time of the current moving object. For example, if too many cars are moving on the same road at the same time, all cars must reduce their speeds, or some cars should change lanes to other less trafficked lanes. By applying typical laws on moving objects, the authors claimed that their prediction of the location of moving objects are more accurate than that of the earlier approach [13].

Considerable transportation network modeling (e.g., number of lanes, lane directions, and traffic light positions) is available from the TRANSIMS [8] software package. useful predictions will need to account for realistic road networks. The TRANSIMS model is probably a good starting point.

### 4.2. What happens when an object moves from one polyline to another during the time interval [t1, t2]?

Typically, when an object *oid* moves from one polyline to another during the time interval [t1, t2], the object will be updated to move only on the new road [4][13]. For example, at a time t1'=t1+$\Delta t$, the object begins to go on a new road. Its new time interval [t1', t2] and its new position interval on the road will be updated in the tree. In our opinion, the prediction about the movements of objects will be more realistic if we

split the time interval into two time intervals [t1, t1'] and [t1', t2]. We then compute the corresponding position intervals of these two time intervals. This object *oid* will be inserted into the tree two times along with its two different polylines and time intervals.

### 4.3. Why is the search time of MON-tree worse than that of R*-tree?

From the experiment in Section 3.2, we see that the search time of the MON-tree worse than that of the R*-tree. There are two main reasons. First, the number of nodes in the MON-tree is 1.08 to 1.22 times greater than that of the R*-tree. Second, for searching in the R*-tree, we only need to visit nodes in a tree. The MBB of each node in the R*-tree is compared to one query rectangle at a time, which decide whether we stop or continue traveling down to the deeper levels of the tree. However, in the MON-tree, after reaching each leaf node of the top tree, we have to break the window query into several window sub-queries. Consequently, each node in the corresponding bottom tree of the top tree's leaf will be compared to a set of the sub-queries. If there are many roads falling in or intersecting with the window query, the set of sub-queries will be very large, which increases the search time for the MON-tree.

### 4.4. Is there any new index structure that can improve the search time?

Since the R*-tree for MBBs of polylines runs faster than the MON-tree in searching, we can combine such R*-tree and the strip trees for representing polylines. The R*-tree will be used as the top tree, while each of its leaf nodes is linked to a strip tree, carried the detail of the intersections between the window query and the current polyline.

## 5. Conclusion

Indexing moving objects (e.g., moving vehicles, moving airplanes, moving ships) is one of the open problems of spatio-temporal data structures in recent years. Some work has been done to index moving objects in both a constrained network (e.g., movements of cars on a road network) and a non-constrained network (e.g., movements of ships on ocean) using data structures, such as the R*-tree [2] and TPR-tree [11]. However, for objects moving in a constrained network, a better approach is to limit the movement of objects on the constrained network such as roads or lanes. By doing this, we can predict their velocities and positions more accurately. Based on this idea, the FNR-tree and MON-tree are two new index structures for moving objects on constrained networks. In [5], authors claimed that the times for building the MON-tree and searching on it can be four times faster than those of the FNR-tree.

In order to understand the data structure of the MON-tree thoroughly, this project implemented the MON-tree data structure and compared it to the R*-tree data structure for moving objects. The MON-tree is an interesting data structure for moving objects in constrained networks. It is constructed by a hash table, a top R-tree, and a set of bottom R-trees linked to the top R-tree and the hash table. The experiment in Section 3.2 shows that the running time for building trees of the MON-tree is faster than that of the R*-tree. However, the MON-tree 's average search time is worse than that of the R*-tree. Also, when the number of objects falling in range is large, the average search time of the MON-tree is also worse than that of the naïve search.

# References


[1] The web page: Canadian road network. http://geodepot.statcan.ca/Diss /Data/Datae.cfm, Last accessed: December 18, 2006.

[2] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD Conference*, pages 322–331, Atlantic City, New Jersey, United States, May 23-26, 1990.

[3] Martin Breunig, Can Türker, Michael H. Böhlen, Stefan Dieker, Ralf Hartmut Güting, Christian S. Jensen, Lukas Relly, Philippe Rigaux, Hans-Jörg Schek, and Michel Scholl. Architectures and implementations of spatio-temporal database management systems. In *Spatio-Temporal Databases: The CHOROCHRONOS Approach*, pages 263–318, 2003.

[4] Jidong Chen, Xiaofeng Meng, Yanyan Guo, Stephane Grumbach, and Hui Sun. Modeling and predicting future trajectories of moving objects in a constrained network. In *The seventh International Conference on Mobile Data Management*, volume 0, pages 156–163, IEEE Computer Society, Los Alamitos, CA, USA, May 10 - 12, 2006.

[5] Victor Teixeira de Almeida and Ralf Hartmut Gting. Indexing the trajectories of moving objects in networks. *Journal GeoInformatica*, 9(1):33–60, 2005.

[6] Elias Frentzos. Indexing objects moving on fixed networks. In *Proceedings of the 8th International Symposium on Spatial and Temporal Databases*, pages 289–305, Santorini, Greece, July 24-27, 2003.

[7] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, Boston, Massachusetts, June 18 - 21, 1984.

[8] Los Alamos National Laboratory. Transims: The transportation analysis simulation system. University of California, US. http://transims.tsasa.lanl.gov/, last accessed: January 9, 2007.

[9] Philippe Rigaux, Michel Scholl, and Agnès Voisard. *Introduction to Spatial Databases: Applications to GIS*. Morgan Kaufmann, 2000.

[10] John F. Roddick, Max J. Egenhofer, Erik G. Hoel, Dimitris Papadias, and Betty

Salzberg. Spatial, temporal and spatio-temporal databases - hot issues and directions for phd research. *SIGMOD Record*, 33(2):126–131, 2004.

[11] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD Conference*, pages 331–342, Dallas, Texas, United States, May 15 - 18, 2000.

[12] Qingxiu Shi. Source code of the r*-tree for rectangles. University of New Brunswick, Faculty of Computer Science, 2006.

[13] Yufei Tao, Dimitris Papadias, and Jimeng Sun. The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In *The twenty-ninth VLDB Conference*, pages 790–801, Berlin, Germany, September 9-12, 2003.

# Appendix

We used test data from the Canadian road network [1]. The data are digital representation of Canada's national road network, containing information such as street name, type, direction and address ranges. We downloaded the road data of New Brunswick. The format of the file containing these data is GML (Geography Markup Language). The size of this file is 46,145 KB. We then transformed this GML file into a smaller text file, which contains only the coordinates of polylines (i.e., roads). Figure 6 shows an example of the input GML file.

After being transformed, the text file data is used in our testings. The size of the text file is 16,840KB. The corresponding test file data of the GML data in Figure 13 is shown as follows:

-63.78544312566555, 46.13002349989367
-63.785318416885985, 46.1303573552171
-63.78440400663378, 46.1316057113912
-63.78036082388033, 46.134045437308345
-63.77902236753822, 46.13461253836588:

```
<gml:featureMember>
   <GEO:RoadSegment fid="GEO_RT_1096024">
      <GEO:ngd_id>1213269</GEO:ngd_id>
      <GEO:class_>ST</GEO:class_>
      <GEO:addrFmLeft>0</GEO:addrFmLeft>
      <GEO:addrToLeft>0</GEO:addrToLeft>
      <GEO:addrFmRght>0</GEO:addrFmRght>
      <GEO:addrToRght>0</GEO:addrToRght>
      <GEO:centreline>
         <gml:LineString>
            <gml:coordinates decimal="." cs="," ts=" ">-
            63.78544312566555,46.13002349989367 -
            63.785318416885985,46.1303573552171 -
            63.78440400663378,46.1316057113912 -
            63.78036082388033,46.134045437308345 -
            63.77902236753822,46.13461253836588
            </gml:coordinates>
          </gml:LineString>
      </GEO:centreline>
   </GEO:RoadSegment>
</gml:featureMember>
```

Figure 13: Sample of the input GML file.